Improving Hadoop MapReduce Performance Using Speculative Execution Strategy in a Heterogeneous Environment

Zar Zar Oo , Sabai Phyu
University of Computer Studies, Yangon
zarzaroo@ucsy.edu.mm, sabaiphyu@ucsy.edu.mm

Abstract

MapReduce is currently a parallel computing framework for distributed processing of large-scale data intensive application. The most important performance metric is job execution time but it can be seriously impacted by straggler machines. Speculative execution is a common approach for this problem by backing up slow tasks on alternative machines. Some schedulers with speculative execution have been proposed but they have some weaknesses:(i) they cannot calculate the progress rate accurately because the progress scores of the phases are set to constant values which may be totally different for heterogeneous environment, (ii) they define the stragglers by specifying a static threshold value which calculates the temporal difference between an individual task and the average task progression. To get the better performance, this paper proposes an algorithm identifying the stragglers by the more accurate progress of each job based on its own historical information and using a dynamic threshold value adjusting the continuously varying environment automatically.

1. Introduction

The Apache Hadoop, an open-source implementation of MapReduce has been widely recognized and applied with high degree of reliability, extensibility, effectiveness and fault tolerance. One significant characteristic of Hadoop is fault-tolerance. If a task performs poorly on a node, a straggler, Hadoop would re-execute the task on another node to finish the job faster. Without the speculative execution mechanism, the stragglers will prolong the job's execution time. Hadoop default scheduler is considered for homogeneous environment. So, it cannot find the slow tasks correctly in heterogeneous environment.

In Hadoop culsters, hardware configuration and resource virtualization are different and leads to heterogeneous environment. Therefore, detecting these straggler tasks in such environment is the key topic of the research. Some schedulers with speculative execution have been proposed but they cannot detect straggler tasks correctly because they compute the progress of tasks in static manner.

In this paper, the system is proposed to overcome the above limitations. Some important contributions are also presented. The proposed system:

- estimates the progress rate of the running task accurately by using historical information on every node.
- calculates the dynamic threshold value to find the slow tasks effectively.
- classifies slow nodes into map slow nodes and reduce slow nodes.

The rest of this paper is organized as follows. Section 2 provides the background information; Section 3 surveys the related work: Section 4 describes the details of the proposed system; Section 5 discusses conclusions and future work.

2. Background Theory

In this section, the operating principles of MapReduce and Hadoop platform are described and the long tail behaviour in distributed system is overviewed.

2.1. Operating Principles of MapReduce

Hadoop is the most widely adopted open-source implementation of MapReduce framework which is created by Google.

MapReduce plays a critical role for executing highly parallelizable and distributed algorithms for huge data sets across large clusters of hardware nodes. Moreover, it also meets multiple quality requirements by monitoring the order and distribution of users, jobs and tasks execution. It allows the developers to write programs in the programming model which originates from two functions. One is a map function that processes a key/value pair to generate a set of intermediate key/value pairs. Another is a reduce function that merges all intermediate values associated with the same intermediate key.

Many real world tasks are representable in this model. Programs are written in functional style and can be executed parallel on a large cluster of commodity machines [1].

In a MapReduce cluster, the input files for the submitted job is divided into multiple map tasks, and then both the map tasks and reduce tasks are

scheduled to worker nodes. A worker node runs tasks and keeps the task's progress to the master with the periodic heartbeat. Map tasks extract key-value pairs from the input, transfer them to some user defined map and combine function to generate the intermediate outputs. After the reduce tasks copy the map outputs, reduce tasks merge these pieces to a single ordered pair stream by a merge sort. These stream pairs are transferred to user defined reduce function. Finally the reduce task generates the result for the job [4].

In MapReduce execution system, a map task is divided into map and combine phases and a reduce task is into three phases(copy, sort and reduce) as shown in figure 1. Therefore, the progress score of a task is obtained by combining the progress score of every phase.

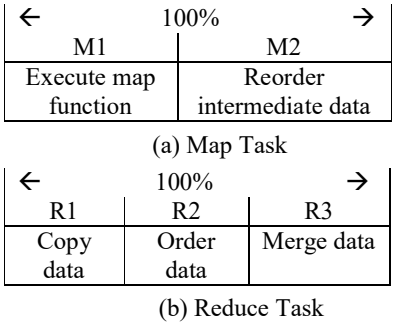

Figure 1. Two phases of a map task and three phases of a reduce task

The progress of a phase can be calculated as in equation (1). *M* is the number of key/value pairs that have been processed in the phase and *N* is the overall number of key/value pairs needed to be processed in the phase. For a reduce task, if the first *K* phases has finished, since each phase occupies 1/3 of the progress score, *PS* is calculated by adding the progress score of the finished phases and the progress score of the current phase.

$$PS[i] = \begin{cases} \frac{M}{N} & For\ map\ task, \\ \frac{1}{3} * \left(K + \frac{M}{N}\right) & For\ reduce\ task \end{cases} \quad \text{.....(1)}$$

where, $K \in (0,1,2)$

$$PS_{avg} = \sum_{i=1}^{n} PS[i]/n \quad \text{.....(2)}$$

For n running tasks, the average progress score is denoted by PS_{avg} which is calculated as in (2).

2.2. Long Tail Behaviour in Distributed System

In MapReduce model, a job is not completed until all data is processed completely; the execution time of the MapReduce job is decided by the last finished tasks, straggler tasks. But it is not a serious problem in homogeneous environments because the nodes in this environment execute tasks with the same data set size in similar time. In heterogeneous environments, the job execution time will be prolonged by the straggler tasks seriously since workers require various time in accomplishing even the same tasks due to their differences, such as capacities of computation and communication, architectures, and memories [3].

Service response time is a vital issue in pay-by-the-hour environments, Amazon EC2, and systems which require rapid response to users. Therefore, this important requirement becomes increasingly challenging in large-scale systems because of the Long Tail in distributed systems [2].

The causes for stragglers can be categorized as shown in table 1.

Table 1. Classification for straggler root-cause

Type	Category	Occurrence frequency
1	High CPU utilization	30%
2	High disk utilization	23%
3	Unhandled operational access request	23%
4	Network package loss	13%
5	Hardware faults	7%
6	Data skew	3%

One of the reasons becoming slow tasks is high CPU utilization which is occurred due to low time-slice sharing and process scheduling due to certain bad user-defined worker logic, unbalanced workload aggregation, etc. High disk utilization is due to local disk read and write conflicts, unbalanced tasks aggregation, disk faults, etc. Distributed file system request surging (usually read request) and overpass the capability of request handling leads to unhandled operational access request.

When network traffic loses a package, repeating intermediate file and data transmission may be resulted. Because of hardware faults, stragglers can occur with server timing-out, hang, etc. Uneven file block input resulting in data skew is one of factors for the occurrence of stragglers.

Stragglers significantly extend job execution time, thus impacting QoS and consumer Service Level Agreement (SLA). Even unusual performance abnormalities can affect a significant portion of all requests in large-scale distributed systems. As a result, the detecting and solving long tail challenge is critical in order to speed up job completion and enhance operational efficiency of heterogeneous system performance.

There are two approaches to mitigate stragglers; avoidance and tolerance. Avoidance occurs within the task scheduling phase and tolerance is in run-time phase. To avoid the stragglers, schedulers assign map tasks to a node with data locality overcoming the unnecessary network transmission overhead. They try not to schedule tasks on faulty nodes by adopting blacklist techniques. However, blacklisting may be unsatisfactory when stragglers are not restricted to a small set of machines.

For most of these factors, straggler tolerance is the most commonly applied method for speculative execution. As high CPU utilization and high disk utilization, the resource competition due to co-hosted applications cannot be avoided since it is not impossible to control over other users' VMs. Among these root-causes, the resource capacity heterogeneity of worker nodes is usually steady and predictable. For most of these factors, speculative execution is an effective way to solve the long tail problem.

2.3. Speculative Execution

When Hadoop framework encounters that a certain task is taking longer on average compared to the other tasks from the same job, it clones that task and runs it on another node. This is called Speculative Execution [6]. After completion of a task, all running duplicate tasks are killed.

Detecting the real straggler tasks to duplicate is very important because the remaining time of all the tasks can be estimated wrongly. The mistaken detected straggler tasks can cause at least two problems. First, the performance of the MapReduce job cannot be upgraded because of the wrong straggler tasks since the real straggler tasks still prolong the job execution time. Second, the backup tasks for the wrong straggler tasks also waste system resources. The disputation on the system resources even degrades the overall performance of the MapReduce job.

In speculative execution, two policies: the least progress policy and the longest remaining time policy are used for detecting the slow tasks. Some speculative execution based Hadoop MapReduce schedulers are analysed in next section.

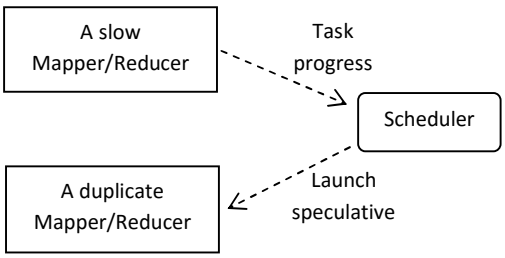

Figure 2. Speculative Execution

3. Related Work

Several speculative execution strategies have been proposed in the literature, including MapReduce in Google [3], Hadoop [7], LATE [2], Dryad in Microsoft [8], SAMR [9], ESAMR [10].

The original MapReduce implementation in Google and Dryad use the same speculative execution mechanism. When the tasks are close to completion, the remaining tasks are selected arbitrarily to backup as long as slots are available. This strategy is very simple and natural. However, they do not consider whether the remaining task is really slow or if they have more data to process. They never consider whether the worker node chosen to run is fast or not and if the backup task could complete before the original task.

Hadoop default scheduler improves this mechanism by using the progress of a task and starts the speculative execution when a job has no new task to assign. It identifies a straggler and subsequently launches a replica based on progress score of task execution as shown in equation (3).

For a task k,

$$PS[k] < PS_{avg} * 80\% \quad \text{.....(3)}$$

, where PS_{avg} is the average progress score of a job and n is the number of tasks being executed.

Longest Approximation Time to End (LATE) algorithm is robust to node heterogeneity, because it will launch only the slowest tasks. It monitors the progress rate of tasks and estimates their remaining time. It selects the tasks as backup candidates when task's progress rate is below *slowTaskThreshold*. Among the backup tasks, the task with the longest remaining time is given the highest priority to be backed up. It tries to launch the speculative tasks on fast nodes. However, the truth of the remaining time calculation depends on the weight of each phase, which is static, without considering the characteristics of different tasks. Therefore, LATE's estimated result may differ from the actual value.

Self-Adaptive MapReduce Scheduling Algorithm (SAMR) updates dynamically the phase weights based on historical information improving the accuracy of the estimation of task's remaining execution time. However, it assumes that there is only a single job executing in the system. SAMR has also a problem because the actual phase weights of the jobs will be different while multiple jobs are executing in parallel.

Enhanced Self-Adaptive MapReduce Scheduling Algorithm (ESAMR) was proposed to identify the slow tasks accurately. By using k-means clustering algorithm, it differentiates the historical stage weights information on each node and divides them into K clusters. While executing, it classifies the tasks into one of the clusters and uses the clusters weights to estimate the execution time of the job's tasks on the node. ESAMR leads to the smallest error in estimation and identifies slow tasks most

accurately but it is limited to only K-means clustering algorithm.

To overcome the shortcomings mentioned above, the proposed system improves the selecting accuracy of stragglers among multiple jobs executing simultaneously. Similar to SAMR and ESAMR, it calculates the task progress rate by using the historical information on each node to tune parameters. Different from them, the proposed system considers the fact that the different types of jobs may have different phase weights during multiple types of jobs executing. The above schedulers configured a default threshold value of 80% which is an unavoidable limitation where tasks that have completed more than 80% progress can never be speculatively executed. To alleviate this limitation, the dynamic threshold value is calculated in the proposed system as illustrated in section 4.4.

4. The Proposed System

In this section, the environment heterogeneity aware scheduling algorithm is proposed.

Algorithm 1 The Runtime algorithm
Input: Key/Value pairs
Output: Statistical results

Initialize the scheduler:
Step1: Reading historical information and adjusting parameters by using the self-learning strategy on every worker node as shown in section 4.2.

Process tasks:
Step2: Computing the progress scores of all the running tasks on every worker node by using the progress monitoring algorithm and the dynamic threshold calculating algorithm as shown in section 4.3.
Step3: Processing tasks, detecting straggler tasks using the straggler detecting algorithm as shown in section 4.4.
Step4: Detecting slow nodes using the slow node detecting algorithm as shown in section 4.5.
Step5: Beginning backup tasks on a suitable node which is determined by step 4.

Termination: Collecting results and updates historical information on every node.

4.1. Overview of the Proposed System

Algorithm 1 lists the runtime algorithm of the proposed system. It will detect straggler tasks based on the accurate progress score and achieve better performance. Starting to execute a MapReduce job, from the local node, each worker reads historical information which contains the values of *M1, M2, R1, R2* and *R3*. Based on these dynamic-tuned values, the proposed system will compute the progress score of tasks more accurately, detect the straggler tasks and classify slow nodes by using the average progress rate of map tasks and reduce tasks on every node. During monitoring the stragglers, the dynamic threshold will be used. If there are any straggler tasks, backup tasks will be launched to fast worker nodes. After all input data has been processed, the proposed system will terminate the MapReduce job and report the final result.

4.2. The Self-Learning Strategy

On each worker node, *M1, M2, R1, R2* and *R3* may be different because the task processing speeds can't be same on different nodes. To get the accurate process speed for each worker, the self-learning strategy adjusts the values of *M1, M2, R1, R2* and *R3* dynamically. In this strategy, these values are read from the corresponding node. The default values are used for the first time. Whenever a map task finishes on each worker node, the values of *M1* and *M2* are updated with the new ones. After a reduce task, *R1, R2* and *R3* are also updated [11]. After a task, the new weight value wnew is obtained by (4) where wold is the last recorded value and wfinished is the recently finished task. The value of *TH* ranges from 0 to 1. If *TH* closes to 1, wnew mostly depends on wold and wnew cannot reflect up-to-date features of the current running task. On the other hand, if *TH* closes to 0, the new weight value may be destroyed by random factors, since $w_{finished}$ is likely to be influenced by random events. The best parameter for *TH* is chosen after a series of experiments.

$$w_{new} = w_{old} * TH + w_{finished} * (1 - TH) \dots (4)$$

Since a worker reads and updates historical information from local node, there is not any additional communication. This condition leads to be a scalable system.

4.3. The Progress Monitoring Algorithm and the Dynamic Threshold Calculating Algorithm

While a MapReduce task is being executed, the progress scores of all the running tasks are computed periodically. Suppose the running task θ has been finished K phases and the progress score is computed as in (5) and (6). PS_{phase} has been got by (1).

$$For\ map\ task\ \theta,$$
$$PS_\theta = \begin{Bmatrix} M1 * PS_{phase} & if\ K = 0, \\ M1 + M2 * PS_{phase} & if\ K = 1. \end{Bmatrix} \dots\dots(5)$$

$$For\ reduce\ task\ \theta,$$
$$PS_\theta = \begin{Bmatrix} R1 * PS_{phase} & if\ K = 0, \\ R1 + R2 * PS_{phase} & if\ K = 1, \\ R1 + R2 + R3 * PS_{phase} & if\ K = 2. \end{Bmatrix} \dots\dots(6)$$

Progress score is used to measure the task progress rate (*PR*) which is calculated by (7).

$$PR[i] = PS[i]/T, \quad \ldots\ldots(7)$$

where T is the amount of time that task θ has been executed.

Using the progress rate addresses the limitations of progress score based methods. However, it still comes with its own limitations. For example, there are two tasks: A and B which has faster progress rate but B is only at 10% of its execution lifecycle. At this time, a progress rate based threshold would detect task A as a straggler due to its slower progress than B. However, in reality, task B will significantly delay total job completion time. To solve this limitation, LATE uses the estimated finish time based threshold to calculate the estimated time to completion [2].

$$TTE[i] = (1 - PS[i])/PR[i] \quad \ldots..(8)$$

Among the tasks of the same job, a task is defined as a straggler task when Time To End (*TTE*) value is longer than a certain percentage compared to the average value in the same job. In the proposed system, estimated finish time based threshold only speculatively executes the backup task which will improve job response time. In this paper, this time threshold is the primary type of focused threshold for enhancing performance.

The previous works used a pre-defined value, a static time threshold value which can decline the efficiency of speculative replica generation. The resource contention for launching a backup is ignored in previous works. This may be a negative impact within the system. To be effective time threshold calculation method, the essential diversity of job timing constraints should be considered. The proposed system also considers the ability to execute different levels of severity for a backup creation to organize with the specified levels of QoS [12].

The proposed system also defines a dynamic time threshold which indicates when a backup task should be created for accepting task stragglers by considering three key features: QoS timing constraints, task progress and system resource usage. The dynamic threshold calculator for task i at time interval tj is

$$Th_j = Q_j + \delta * P_j + \theta * R_j \quad \ldots..(9)$$

where Q calculates the difference between a task's estimated completion time with respect to specified job QoS timing requirement at time interval t_j in (10). P represents the optimal backup tasks creation according to task lifecycle as in (11). R determines the current resource utilization level of the heterogeneous system as in (12). System administrator can specify the process weight parameter δ and utilization weight θ to optimize the trade-off for replica creation based on specific system operation goals. At the time of greater value Th_j, it means that fewer tasks will be defined as stragglers while a lower Th_j value allows a more comfortable condition for creating speculated replicas.

4.3.1. The QoS influence

QoS timing constraint is an important factor to be considered when deciding how severe the time threshold should be based on the nature of the application.

$$Q_j = \begin{Bmatrix} \dfrac{QoS}{medium(TTE_i)}, QoS \geq \max(TTE_i) \\ \dfrac{\min(TTE_i \ \ larger\ than\ QoS)}{medium(TTE_i)}, else \end{Bmatrix} \quad \ldots(10)$$

where TTE_i represents the estimated time to completion for the i^{th} task within the cluster, and QoS is the request time requirement.

4.3.2. Progress Adjustor (P_j)

$$P_j = \frac{\sum_{i=1}^{n} PS[i]}{n} - \mu \quad \ldots.(11)$$

PS[i] value ranges from 0 to 1, the start and the end of $task_i$ respectively. μ is the standard parameter that represents the specified maximum point within a task's lifecycle suitable for generating a replica.

4.3.3. System Environment Adjustor (R_j)

$$R_j = \max\left(\frac{\sum_{i=0}^{n} CPU_{req_i}}{n} - \alpha, \ \frac{\sum_{i=0}^{n} MEM_{req_i}}{n} - \beta\right) \quad \ldots.(12)$$

R_j affects the threshold value which depends on the system utilization at t_j. The parameter n denotes the number of servers within the cluster system. CPU_{req} and MEM_{req} represent the CPU and memory requirement of $task_i$ and α, β are standard thresholds respectively. By using the above three parameters, the required dynamic threshold value is calculated as in (9).

4.4. The Straggler Detecting Algorithm

If the $task_i$ satisfies the equation (13), it is defined as a straggler task. After that, a replica is created and will launch execution.

$$TTE[i] > Th_j * TTE_{avg} \quad \ldots..(13)$$

4.5. The Slow Node Detecting Algorithm

To detect slow nodes in the system, the average progress rate of the running map/reduce tasks on a node is used. For a node ϕ with M map tasks and R reduce tasks,

$$MR_{\emptyset} = \frac{\sum_{i=1}^{M} PR_i}{M}, \ PR_{\emptyset} = \frac{\sum_{i=1}^{R} PR_i}{R} \quad \ldots\ldots(14)$$

For node ϕ, if

$MR_{\emptyset} < (1 - Node_{cap}) * MR_{avg}$, it is a map slow node.

If $PR_{\emptyset} < (1 - Node_{cap}) * PR_{avg}$, it is a reduce slow node. MR_{avg} and PR_{avg} are the average map/reduce tasks progress rate of all the nodes.

$Node_{cap}$ is a cap of slow proportion of the slow node. Since the proposed system has detected out straggler tasks and map/reduce slow nodes, it can simply allocate backup tasks on the appropriate worker node.

5. Conclusion and Future Work

Traditional MapReduce schedulers cannot identify slow tasks correctly because the progress scores of tasks are estimated based on inaccurate weight of each phase in the overall progress of a task. The proposed system uses the record of the previous works to detect the stragglers accurately and classifies slow nodes into map slow nodes and reduce slow nodes. Moreover, a dynamic threshold captures job QoS, system resource usage level, and task progress. So, it leads to achieve better performance and the dynamic time threshold can improve job completion, decrease timing failure occurrence and save resource under high utilization scenarios by generating fewer replicas. To evaluate the performance of the proposed system, a series of simulation experiments will be accompanied using distributed system simulator. The future work is to launch backup tasks on nodes with the corresponding data set of the straggler task.

References

[1] H. Li, Introduction to Big Data, New York, October 31, 2015.

[2] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments", 8th USENIX Symposium on Operating Systems Design and Implementation, March 2009, pp. 29-42.

[3] J. Dean and S. Ghemawat "MapReduce: simplified data processing on large clusters", Communications of the ACM, vol.51, January 2008, pp.107-113.

[4] Q. Chen, C. Liu, and Z. Xiao, "Improving MapReduce Performance Using Smart Speculative Execution Strategy", IEEE Transactions on Computers, Volume 63, Issue 4, April 2014, pp. 1-14.

[5] P. Garraghan, X. Ouyang, R. Yang, D. McKee, J. Xu, "Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters", IEEE Transactions on Services Computing, 2016, pp. 1-13.

[6] Speculative Execution. [Online]. Available: http://hadoopinrealworld.com/speculative-execution/

[7] "Apache hadoop, http://hadoop.apache.org/".

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, ser. EuroSys '07, 2007.

[9] Q. Chen, M. Guo, Q. Deng, L. Zheng, S. Guo, Y. Shen, "SAMR: A Self-adaptive MapReduce Scheduling Algorithm In Heterogeneous Environment", 10th IEEE International Conference on Computer and Information Technology, 2010, pp. 2736-2743.

[10] Xiaoyu Sun, "An Enhanced Self-adaptive MapReduce Scheduling Algorithm", Master Thesis, University of Nebraska, Lincoln, 2012.

[11] Q. Chen, M. Guo, Q. Deng, L. Zheng, S. Guo, Y. Shen, "HAT: history-based auto-tuning MapReduce in heterogeneous environments", The Journal of Supercomputing, June 2013, Volume 64, Issue 3, pp 1038–1054.

[12] X. Ouyang, P. Garraghan, D. Mckee, P. Townend, J. Xu, "Straggler Detection in Parallel Computing Systems through Dynamic Threshold Calculation", 30th IEEE International Conference on Advanced Information Networking and Applications", 2016, pp.414-421.